

The Design Environment

Stephan Clemens, Marcello La Rosa and Arthur ter Hofstede

1 Introduction

The core modeling component of the YAWL system is the *Editor*. This tool enables workflow designers to graphically define complex YAWL process models, and to analyze and export these models to the Engine.

Specifically, the Editor is a rich client application offering visual support for the definition of the process control logic, the data associated with the process and its tasks, and the organizational resources participating in the process. An important aspect of the Editor is the provision of sophisticated logic to verify the produced models. Through this capability, a designer can pinpoint syntactical and semantic issues at a mouse click, so as to avoid potentially costly mistakes before deploying a process to a workflow engine.

The main driver behind the development of a visual editor for YAWL was the necessity to speed up the creation of YAWL process models and to foster the uptake of the language by non-technical users. In light of this, the Editor had to fulfill the requirements of *free availability*, *portability*, *ease of use* and *interoperability*. The tool had to be freely available, therefore it was decided to release it under the open source LGPL license. Portability was guaranteed by developing the Editor in Java™ and avoiding any code dependency on OS-specific libraries. Ease of use was achieved by providing the Editor with an intuitive user interface on top of a core

Stephan Clemens
Queensland University of Technology, Brisbane, Australia, e-mail: stephan.clemens@qut.edu.au

Marcello La Rosa
Queensland University of Technology, Brisbane, Australia, e-mail: m.larosa@qut.edu.au

Arthur ter Hofstede
Queensland University of Technology, Brisbane, Australia, e-mail: a.terhofstede@qut.edu.au

graphical component based on extensions to the JGraph libraries¹. Interoperability was supported by the definition of a common XML format and a set of API calls for the exchange of workflow specifications between an editor and the runtime environment (see Chapter ??). The interaction between design and runtime environment in the YAWL system is depicted in Figure 1.

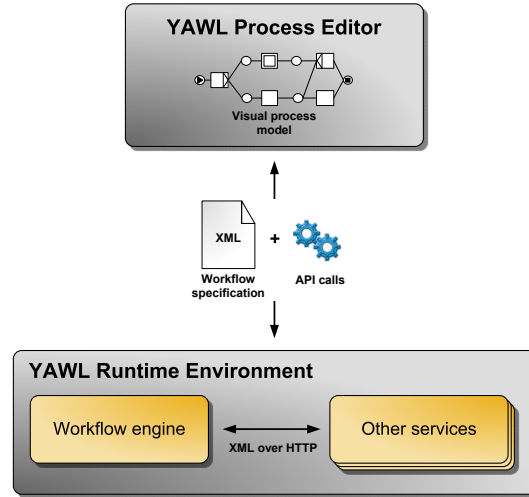


Fig. 1 Workflow specifications created with an editor are deployed to the runtime environment.

The decision to minimize the dependencies between the design environment and the runtime environment via the use of a common interchange format and a set of APIs, is inspired by the Workflow Reference Model proposed by the Workflow Management Coalition (WfMC).² In WfMC terminology, an editor embodies a *Process definition tool* for modeling and analysis of workflows, while the runtime environment represents a *Workflow Enactment Service* for the execution of such workflows. The interaction between these two components is achieved via the *Workflow Definition Interchange* (Interface 1), which is embodied by the XML format and the set of APIs in the YAWL system.

According to the WfMC, the advantage in using a standard interaction format is twofold. Firstly, it defines a clear point of separation between design and runtime environments. Secondly, it enables the use of different design tools and workflow engines. For example, a workflow specification exported by an editor could be executed by a number of independent workflow engines that cooperate to provide a distributed runtime environment. Similarly, it is also conceivable to think of a development environment in which a number of design tools can export workflow

¹ www.jgraph.com

² www.wfmc.org

specifications to the same runtime environment. For example, these tools could differ in their look'n'feel or preferences to address different classes of users.

2 Setting up the Process Control Logic

The Editor user interface comprises a modeling canvas located in the centre, two toolbars – one on top and the other on the left of the canvas, a bottom panel for notes/problems and a status bar reflecting the status of the Engine, Resource Service and general messages. Figure 2 shows a screenshot of the user interface with the starting net *Overall* of the *Order Fulfillment* process model (see Appendix ??) displayed on the canvas.

The control flow of a YAWL specification can be defined by using elements from the palette, located at the top of the left toolbar. The palette *Wf-Elements & Tools* contains seven buttons which assist with creation, selection and positioning of workflow elements on the canvas (see Figure 2). The first five buttons are used to place workflow elements on the net, such as an atomic task or a condition. The marquee button allows the selection of individual or multiple elements which can be moved within the net, while the Net Drag button (indicated by a cross) is used to drag or modify single workflow elements on the canvas.

The palette *Task Icons* contains a set of predefined icons to decorate tasks (see Figure 2). For example, icons can be used to distinguish tasks that are executed by a human resource from those executed by an application, as in the *Order Fulfillment* process model. This set of standard icons can be enriched with custom-made icons, added through a plug-in mechanism. The use of icons can increase the overall understanding of a process, but it will not influence the task's behavior.

A YAWL workflow specification is composed of one starting net and zero or more sub-nets. The starting net captures the behavior of the overall process and is the first net to be executed when a case is launched. Each sub-net captures the behavior of a composite task and is executed once the respective task fires. For example, in the *Order Fulfillment* process model, the net labelled *Overall* is the starting net, while nets *Ordering*, *Payment* and *Freight in Transit* are sub-nets mapped to the homonymous composite tasks (see Figure 2). The division of a process model into smaller parts by means of sub-nets can facilitate maintenance and readability, especially in the case of complex process models such as the *Order Fulfillment* example.

Each net features two mandatory elements: the input and the output condition, which cannot be removed from the canvas. Tasks can be arranged between these two nodes to describe the control flow of the process and are connected to each other by means of arrows, which represent order dependencies.

While a composite task is a placeholder for a sub-net, an atomic task captures a standalone piece of work, namely a work item. There are three types of atomic task: manual, automated and routing task. A manual task is executed by a human resource, e.g. a particular employee or an organization participant with a certain role or position. In the sub-net *Ordering* (Figure ??) of the *Order Fulfillment* process

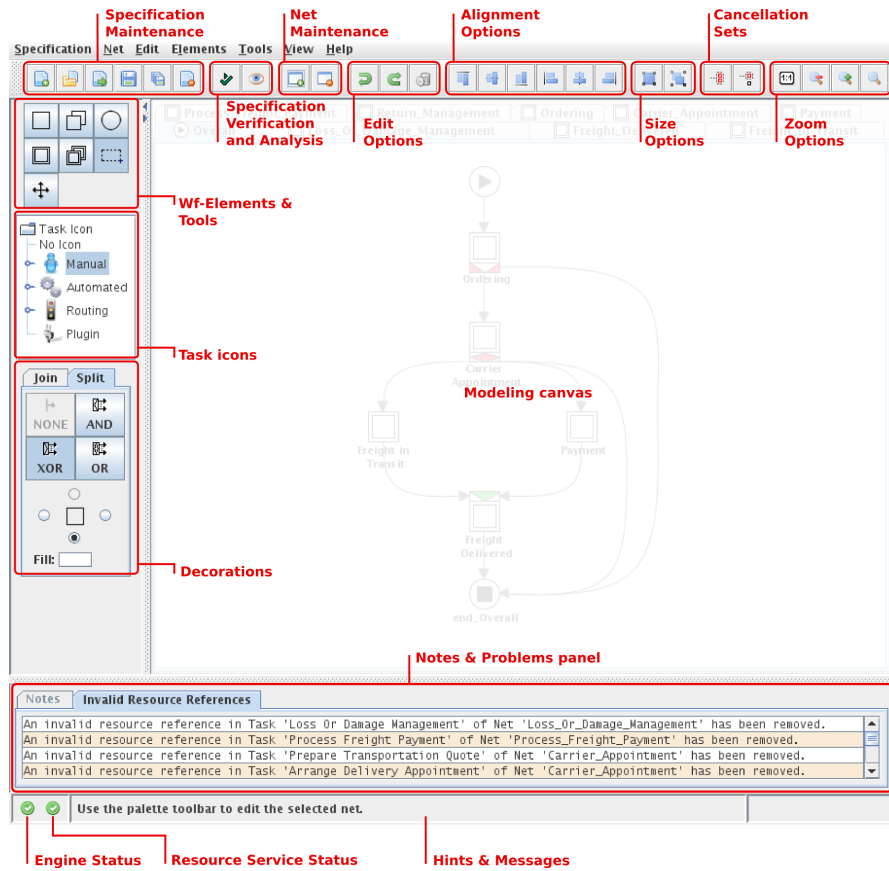


Fig. 2 Graphical user interface of the Editor.

model, tasks *Create Purchase Order* and *Confirm Purchase Order* are manual tasks which are performed by the Purchase Order Manager. The Editor supports the workflow designer in setting up resource related aspects, e.g. which manual task will be performed by which resource or what strategy to use when allocating a work item. This can be done via the *Manage Resourcing...* dialogue which is available from the context menu that pops up by right-clicking on the task. More details on resource assignments are provided in Section 4.

An atomic task associated with the default worklist handler is manual by default. If set as automated, a designer may assign a codelet or an external application to the task, via the *Update Task Decomposition* dialogue available from the task's context menu. Codelets are code snippets that are executed internally by the Engine. To select a codelet the Editor needs to be connected to the Resource Service. A predefined set of codelets is provided with the Editor. These include a codelet for executing shell commands and a codelet for evaluating XQuery expressions. In addition, workflow designers can plug their own codelets into the YAWL environment.

Similarly, in order to assign an external application to an atomic task, the application needs to be exposed as a Web Service and the latter be registered with the YAWL environment. The YAWL system provides a number of custom services already (see Chapter ??). In order to associate a custom service with a task, the Editor needs to be connected to the Engine. If the Engine is active but the status icon in the Editor (see bottom left of Figure 2) states it as offline, a connection can be manually established by invoking the dialogue *Engine Connection Settings* via menu entry *Tools*.

If an atomic task is neither manual nor automated, it is an empty task. An empty task may be a routing task, which is essentially a silent task that is only used for routing purposes. Routing tasks are executed internally by the Engine.

Another type of task is the multiple instance task, which captures an atomic or composite task to be executed multiple times in parallel. The upper and lower bounds for the number of instances, the threshold for completion and the way of instance creation can be specified via the *Set Instance detail...* dialogue available from a multiple instance task's context menu.

Figure 3 shows the tasks hierarchy in the Editor.

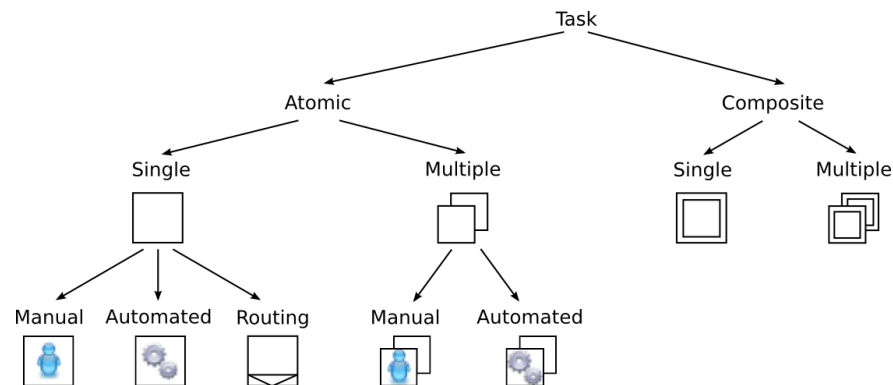


Fig. 3 Tasks hierarchy in the Editor.

An atomic single-instance task can have a timer set if it needs to be executed within a given timeframe. The parameters of a timer task can be specified via the *Set Task Timer...* dialogue accessible from the task's context-menu. These are the expiry value and the activation type. The expiry value indicates when the timeout should expire. This can be after a period of time (e.g. 3 hours) or at a specific point in time (e.g. 25-12-2008). The activation type depends on whether the task is manual or automated. In the case of a manual task, the designer can specify whether the timer should be activated when work items of that task are enabled or when they are started. In the case of automated tasks, the timeout behaves as a delay, i.e. the execution of the task is delayed by the timeout value. Upon timeout expiry, the task is started. In contrast to manual tasks, automated tasks are not assigned to any human resource. Therefore, the timeout is always activated upon work item enablement.

In the *Order Fulfillment* process model the *Ordering* sub-net has an automated timer task *Order Timeout* (see Figure ?? in Appendix ??). This task is set to a duration of 3 days as shown in Figure 4. After the purchase order has been approved it can be either confirmed (task *Confirm Purchase Order*) or modified (task *Modify Purchase Order*). Should none of these two tasks be executed within 3 days, the *Order Timeout* task will then be triggered.

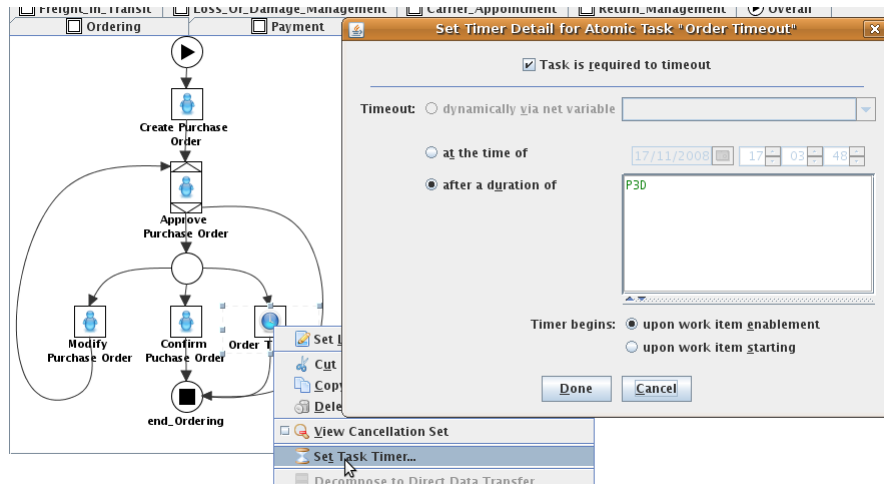


Fig. 4 Setting up the parameters of the timer task *Order Timeout*.

In the case of multiple incoming flows the task needs to be decorated with a join construct, while a split construct needs to be applied in the case of multiple outgoing flows. The type of split and join for a task can be chosen from the *Decorations* palette (see Figure 2), and appears as soon as the task is selected on the canvas. As long as a task has no join or split decoration, the Editor does not allow the connection of more than one incoming or outgoing arc to that task to prevent creating structural issues.

If a task is decorated with an (X)OR split, the workflow designer can specify routing conditions for each outgoing arc of the split. These conditions are predicates expressed in XPath, which can be set via the *Update flow detail...* dialogue available from the task's context-menu (see Figure 5). This dialogue shows the list of outgoing arcs of the split, where each arc is referred to by the label of the subsequent node. In the case of an XOR split, the list of arcs needs to be ordered to determine the evaluation order of the predicates at runtime. Control will be passed along the first arc whose predicate evaluates to true. In this way only one branch will be executed even if more than one predicate evaluates to true. In the case of an OR split, control will be passed along each arc whose predicate evaluation is true. If all predicates of an (X)OR split evaluate to false, control will be passed along the designated default arc (the last arc of the list) irrespective of the predicate's result.

The atomic task *Approve Purchase Order* in the sub-net *Ordering* (Figure ??) is decorated with a split and a join. The XOR construct at the bottom of the task splits the control flow: if the purchase order is not approved the flow is routed to the end-condition of the sub-net. Otherwise the flow is routed to the subsequent condition leading to the atomic tasks *Modify Purchase Order*, *Confirm Purchase Order* and *Order Timeout*.

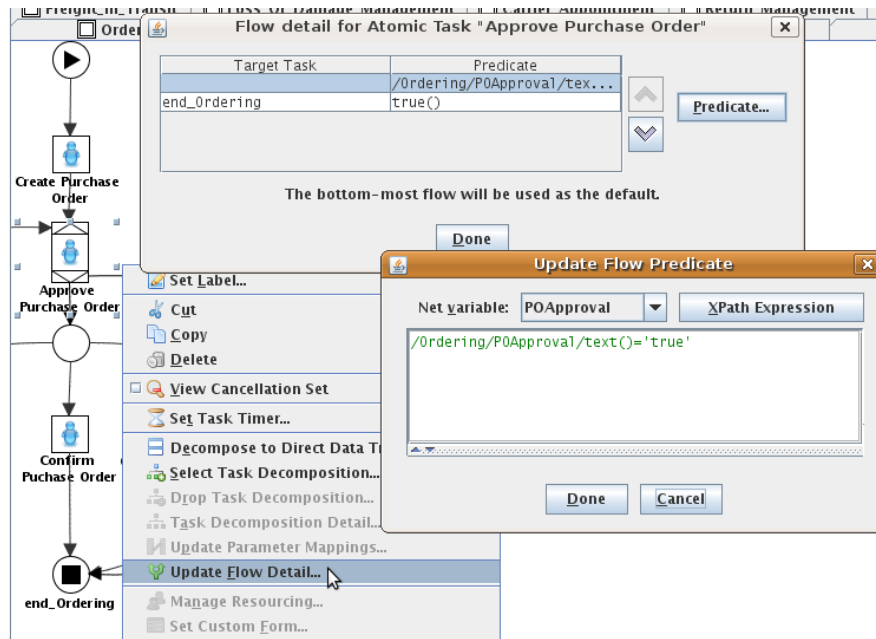


Fig. 5 Setting up routing conditions for an XOR split.

A task can be associated with a cancellation set which may include a number of tasks, conditions and/or arcs (in the latter case the implicit condition in the arc is included). The cancellation set of a task can be visualized on the canvas by clicking on *View cancellation set* from the task's context-menu. This enables two buttons in the top toolbar (*Cancellation Sets* see Figure 2), which can be used to add in elements or remove them from the cancellation set. The task that initiates a cancellation set is grayed out, while the elements in the cancellation set are indicated with a red border.

3 Defining Data Aspects

In a YAWL specification, the data flow is captured by means of net variables. The passage of data to/from tasks is achieved by mapping net variables to tasks' vari-

ables. At runtime, variables are used to compose the content of a task's work item and to determine the routing behavior of splits.

A variable acts as a container in which a value of a certain XML Schema data type is stored. The Editor supports the full set of simple XML Schema types, e.g. *long*, *string*, *boolean*. Complex XML types can be defined as compositions of simple and complex types via the *Update Data Type Definitions* dialogue, which can be found under the *File* menu. For example, *PurchaseOrderType* is a complex type used in the *Order Fulfillment* process model to describe the content of a purchase order (see Figure 6). It consists of further complex type variables such as *Company* and *Order*, and of simple type variables such as *DeliveryLocation* (*string*) and *PrePaid* (*boolean*).

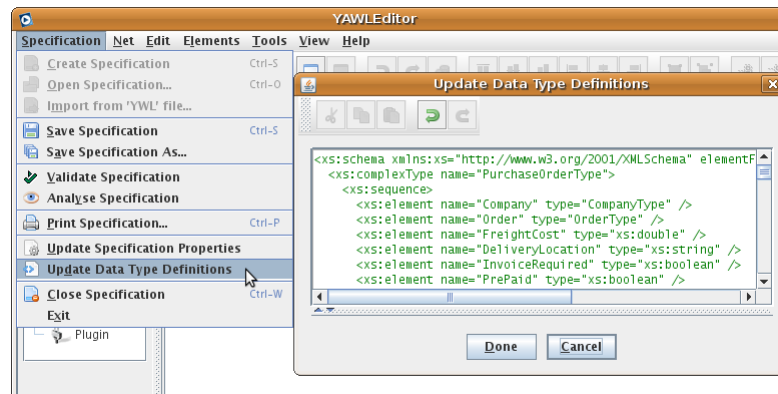


Fig. 6 Complex XML Schema type definition of PurchaseOrderType.

Before nets and tasks are able to read or modify data, they need to be assigned a decomposition (see Chapter ??). A decomposition has one decomposition label (a unique identifier within the whole workflow specification) and may have one or more variables. The Editor facilitates the assignment of the same decomposition to multiple tasks and prevents the workflow designer from creating more than one decomposition with the same label.

The details of a net decomposition (i.e. its label and variables) can be set up through the *Update net detail...* dialogue, under the *Net* menu, whereas task decompositions can be created via the *Select Task Decomposition* dialogue and modified via the *Task Decomposition detail...*, both available from the task's context-menu. Figure 7 shows the decomposition of task *Create_Purchase_Order*. Here variable *PO_Manager* is of simple data type *string* to contain the username of the purchase order manager, while *POrder* has been assigned the complex type *PurchaseOrderType* as defined in Figure 6.

The Editor supports the workflow designer in setting up inbound and outbound mappings to specify how data is transferred between net and task variables. In particular, for each task's decomposition, input parameters are used to transfer the content

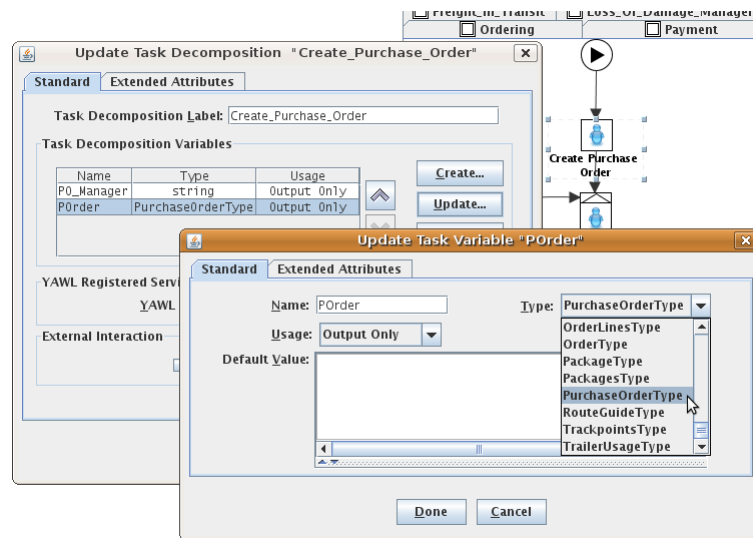


Fig. 7 Decomposition of task *Create_Purchase_Order* and set up of variable *POrder*.

of net variables into task variables (inbound mapping), while output parameters are used to transfer the content of task variables into net variables (outbound mapping). These parameters are defined as XQuery expressions via the *Update Parameters* dialogue from the task's context-menu. Figure 8 shows the mapping for task *Approve Purchase Order* of the sub-net *Ordering* (Figure ??). Here an input parameter is used to copy the content of the variable *POrder* of the net *Ordering* (indicated by the XQuery `/Ordering/POrder/*`) into the task variable *POrder*. Even though both variables are labeled the same they are assigned to different decompositions and therefore different scopes. This copy operation does not create any issue. On the other hand, the two output parameters of task *Approve Purchase Order* take care of copying the content of the task variables *PO_timeout* and *POApproval* to the net variables bearing the same names.

The creation of decompositions can be labor intensive in the case of large workflow specifications. Complex types may need to be created, variables need to be typed and mappings need to be defined between net and task variables. The Editor is able to facilitate this process as long as no decomposition has been assigned to an atomic task yet. This can be done via dialogue *Decomposition to direct data transfer*, accessible from a task's context-menu. Here the designer can use net variables as a template to create a task's decomposition. Specifically, the designer can select a set of net variables to be used as input to the task, and similarly a set of net variables to be used as output from the task. Then the Editor creates task variables bearing the same type of the net variables, input parameters to map input net variables onto task variables, and output parameters to map task variables onto output net variables. Figure 9 shows the direct data mapping for task *Approve Purchase Order*.

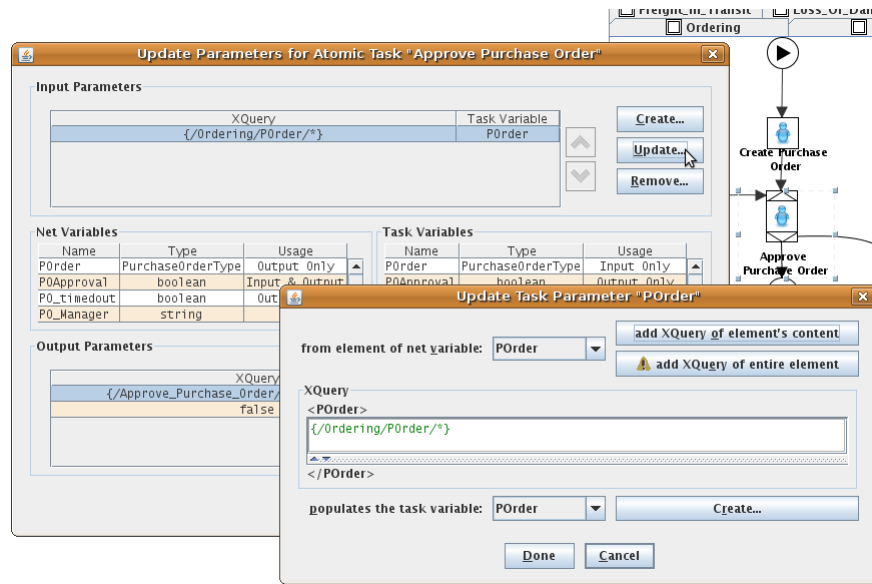


Fig. 8 The input parameter to transfer the content of net variable *POrder* to task variable *POrder*.

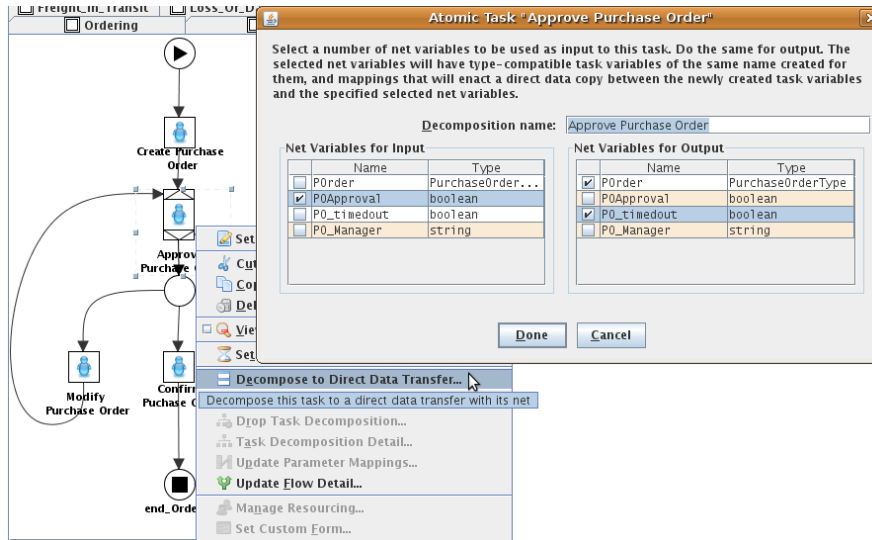


Fig. 9 Decomposition to direct data transfer

Besides input and output parameters, multiple instance tasks need further information for data manipulation. This information is specified via four XQuery expressions. An example of these expressions is provided in Figure 10, which refers to the multiple instance task *Log Trackpoint Order Entry* in the *Order Fulfillment*

example (cf. Figure ?? in Appendix ??). The designer needs to define a *Multiple Instance Variable* that will contain the data to be distributed to the various task instances at runtime. The content of this variable can be taken from net variables using input parameters. The data contained in this variable can be separated with the *Splitter query* to pass a unique value to each instance task. The *Accessor query* can be used to manipulate the content of the variable before the unique values are split out (e.g. if some format conversions are needed). Upon completion of all instances, the *Instance query* can be used to transform the returned XML document from each instance to a form suitable for the *Aggregate query* to generate an overall result. A *Result Net Variable* needs also to be specified to contain the overall result. This variable can then be mapped onto net variables via output parameters.

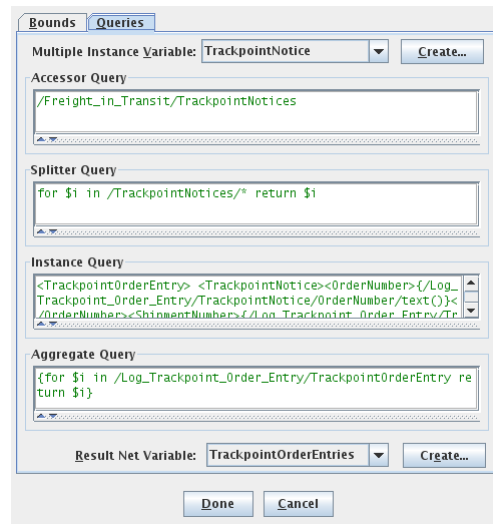


Fig. 10 Four XQuery expressions for a multiple instance task

Finally, data aspects also concern the timer task. This task in fact allows designers to late-bind its expiry value to a period of time or fixed date via the use of a variable of type *YTimerType*. In this way the expiry value will be dynamically determined at runtime.

4 Assigning Human Resources to the Process

The Editor provides a *Resource Manager Wizard* which allows designers to assign participants to manual tasks. This wizard can be invoked by clicking on *Manage Resourcing...* from the task's context menu after participants have been created in the YAWL workflow environment (see Chapter ??), the task has been assigned a

decomposition and the task has been associated with the default worklist (i.e. the Resource Service). It needs to be ensured that the Resource Service is running so that resources can be associated with a task. The status of the Resource Service is reflected by the second right icon in the bottom left corner of Editor (see Figure 2). If the Resource Service is active but the status icon states this service offline, a connection can be manually established by invoking the dialogue *Resource Service Connection Settings* via menu entry *Tools*.

The first step of the wizard allows designers to set up the way a manual task should be offered, allocated and started. These are called interaction points. The interaction points for task *Create Purchase Order* are shown in Figure 11. At the first interaction point the work item gets offered to one or more participants. The offering can be set up as system-initiated or user-initiated. In the case of a system-initiated offering, it is required to specify the group of participants at design time. This can be done in the second and third steps of the wizard.

Step 1 : Choose Behaviour At Interaction Points

There are three key decision points for managing the resourcing of work items spawned from a task. At each of these *interaction points*, you may choose to have the system dynamically make a decision on resourcing at each point, or alternately, allow a user to manually make each decision. Each interaction point is briefly described below:

- **Offer:** The point at which it is decided that a number of participants *could* undertake the work item.
- **Allocation:** The point at which one of the participants offered the work item is *nominated to do* that work item.
- **Start:** The point at which the participant allocated a work item *begins* working on it.

Offering a work item for this task to a number of participants is to be done by: ☐ User ☒ System

Allocating a work item for this task to one of the offered participants is to be done by: ☒ User ☐ System

Starting an allocated work item of this task is to be done by: ☐ User ☒ System

< Back > Next Finish

Fig. 11 Step one: choosing how to offer, allocate and start a work item.

Step two enables a workflow designer to select individual participants by their name, e.g. Peter Clemenza, and/or participants of a certain role, e.g. PO Manager (see Figure 12).

In step three it is possible to restrict the group of participants to those who belong to a certain organizational group or positions or have appropriate capabilities (see Figure 13).

In some cases it may be required to defer the choice of which participant will get offered a particular work item until runtime. This can be done by setting the offering to user-initiated, to allow a user with offering rights (e.g. an administrator) to determine whom the work item is to be offered to at runtime. Alternatively, the

Step 2 : Specify System Behaviour when Offering a Work Item

The offer process involves choosing which participants should be informed of the existence of the work item, one of whom will eventually do this work. As you have specified the system manage the offer process, you must now choose who the work item should be offered to. Begin by specifying a set of participants and/or to distribute offers of work to. You may also specify a net parameter which at runtime will contain a participant's userid or the name of a role.

Participants
 Kay Adams (ka)
 Billy Van Arsdale (bva)
 Momo Barone (mb)
 Emilio Barzini (eb)
 Peter Clemenza (pc)
 Stephanie Clemenza (sc)
 Unselect All

Roles
 Account Manager
 Carrier Admin Officer
 Client Liaison
 Courier
 Finance Officer
 Junior Supply Officer
 Order Fulfillment Manager
 PO Manager
 Senior Finance Officer
 Senior Supply Officer
 Unselect All

Net Parameters

Name	Refers to
------	-----------

< Back > Next Finish

Fig. 12 Step two: specifying the distribution group for work item offering.

Step 3 : Specify Distribution Set Filter(s)

Filters

Select the filters to be applied to the distribution set. Set parameter values for the selected filter as required.

Filter	Parameter	Value
☐ Filter by Organisational Data		
☒ Filter by Capability		

Runtime Constraints

☐ Allow this task to be piled to a single participant.

☐ Choose participant(s) who completed previous task: Timeout

☐ Do not choose participant(s) who completed previous task: Timeout

< Back > Next Finish

Fig. 13 Step three: filtering the distribution group.

Editor enables the use of a variable to store the participant's user identification (late-binding). The variable can be specified within step two of the wizard. In the sub-net *Ordering*, task *Create Purchase Order* allows one to specify a participant who will work on the purchase order if the latter needs to be modified. So, should the task *Modify Purchase Order* be executed, the participant in the net variable *PO Manager* will get offered the work item.

At the second interaction point, the work item gets allocated to a participant out of the group of participants who have been selected for this work item. Once a work item is allocated to a specific participant, the work item offer is withdrawn from all other participants. In the case of a user-initiated allocation, the participant who is offered the work item can decide whether or not to accept it. On the other hand, in a system-initiated allocation the workflow system decides which participant the work item should be allocated to. However, in this case the workflow designer needs to define the appropriate allocation strategy during design time. The fourth step of the Resource Manager Wizard provides several strategies to choose from, e.g. Round Robin or Shortest Queue (see Figure 14).

Fig. 14 Step four: specifying the allocation strategy.

At the third interaction point, the work item gets started (i.e. executed) by a participant. This is usually done via a Web form that is linked to the manual task. The Editor allows a designer to specify the URL of this form by clicking on *Set Custom Form...* from the task's context-menu. If no custom form is specified, a default form is dynamically generated for the workitem at runtime. If the third interaction point is user-initiated, the participant whom the work item has been allocated to can decide when to start the execution of that work item. In the case of a system-initiated start, it is the workflow environment which places the work item in the participant's started queue.

Finally, step five of the Resource Manager Wizard allows the specification of several participant privileges for runtime, such as suspending or skipping the execution of a work item (see Figure 15).

Step 5 : Establish Default User Runtime Privileges for this Task

Can a participant suspend a started work item of this task? ☒ No ☐ Yes

Can a participant reallocate a work item of this task to another participant, resetting state? ☒ No ☐ Yes

Can a participant reallocate a work item of this task to another participant, retaining state? ☒ No ☐ Yes

Can a participant deallocate themselves from a work item of this task? ☒ No ☐ Yes

Can a participant delegate a work item of this task to another participant? ☒ No ☐ Yes

Can a participant skip a work item of this task? ☒ No ☐ Yes

< Back > Next Finish

Fig. 15 Step five: configuring participant's privileges for execution.

5 Error Reporting

The underlying philosophy of the Editor is to support the workflow designer in detecting certain undesirable characteristics in the workflow specification at design time. As shown before, XQuery and XPath expressions are strongly used in YAWL for data handling. However, those expressions are not very intuitive to build and can be complex. An incorrect expression can have undesired side effects during workflow execution. Therefore, each XQuery or XPath expression is verified by the Editor when it is entered. Correct expressions are shown in green font color, incorrect expressions are shown in red. Furthermore, in the case of an incorrect expression a suggestion is provided on how the error can be solved. By validating the specification via button *Validate this Specification* in the top toolbar, a table with listed problems appears in the *Specification Validation Problem* panel, located at the bottom of the Editor interface. The entries show details about inconsistencies that will prevent the specification from running in a workflow engine. For example, by validating the specification in Figure 16 the workflow designer will be informed about the missing inbound mapping for variable *bar* of task *Task A*.

Another error reporting feature of the Editor is the analysis of a specification. This can be triggered via button *Analyze this specification* from the top toolbar (*Specification Verification and Analysis*, see Figure 2). Analysis allows the workflow designer to detect potential behavioral problems of the workflow specification. That is for example, deadlock situations, unnecessary cancellations set members or unnecessary OR joins (see Chapter ?? for further information). Analysis results are reported in the *Specification Analysis Problem* panel, displayed at the bottom of

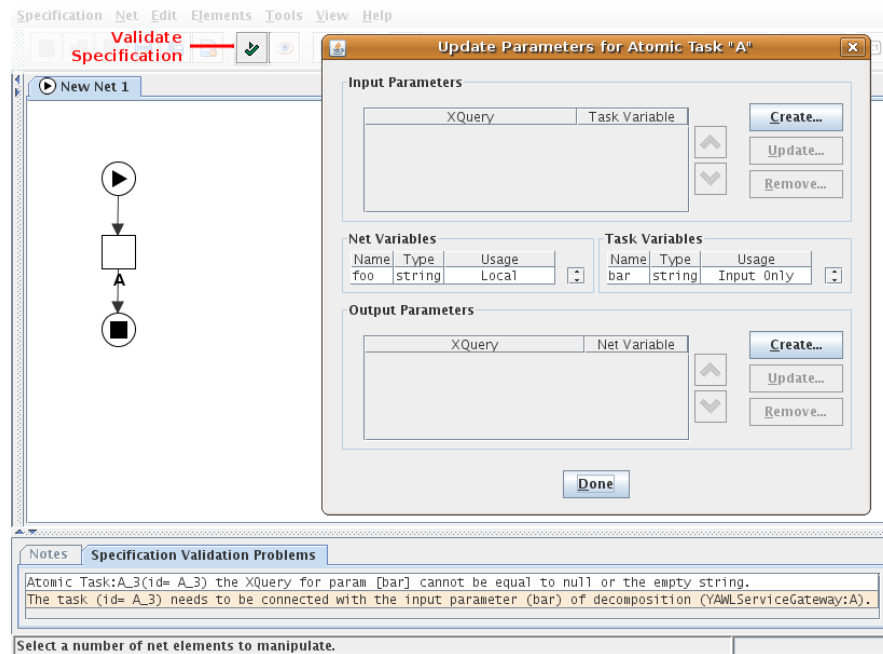


Fig. 16 The missing inbound mapping for variable *bar* causes the listed validation problems.

the Editor interface (see Figure 2). Figure 17 shows a workflow with a potential deadlock and the respective analysis result.

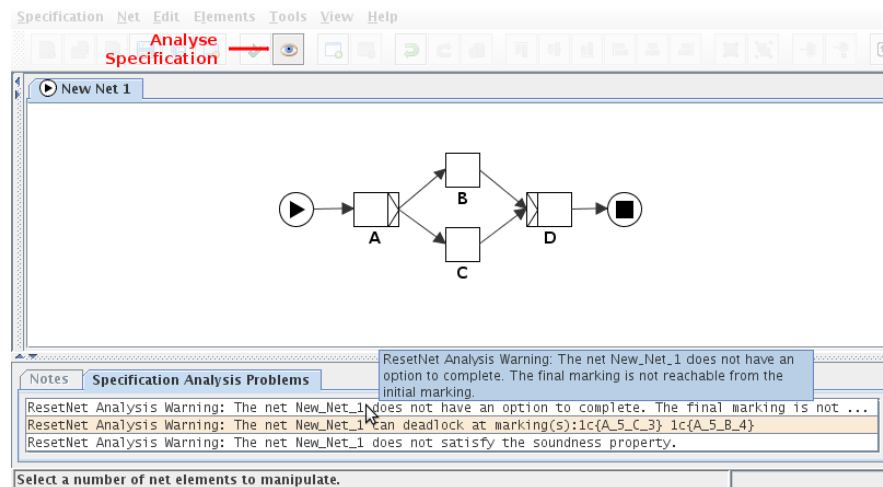


Fig. 17 The Editor detects a potential deadlock.

6 Specification File

The Editor serializes a workflow specification into an XML document with extension `.yaw1`. This document contains a mandatory conceptual part with information about decompositions, workflow elements and their relations.

While the conceptual information is required to execute a workflow, its visualization is important for communication purposes. Workflow elements might be placed in a certain way or enriched with specific icons to improve the process understanding. This information is contained in the layout part, which is optional and ignored by the Engine when a specification is loaded. The Editor uses layout information in conjunction with the conceptual information to represent a workflow. However, if no layout information is enclosed within the XML document, the Editor will visualize the workflow specification with a standard layout.

Listing 1 is an extract of the conceptual information for the *Order Fulfillment* process model (see Appendix ??). It shows a decomposition with its identifier *Create_Purchase_Order* and two of its variables. Below, the task *Create Purchase Order* is represented with its reference to its successor task *Approve Purchase Order*, related mappings and resource information. This task refers to the mentioned decomposition *Create_Purchase_Order*.

```
<decomposition id="Create_Purchase_Order" ...>
  <outputParam>
    <name>PO_Manager</name>
    <type>string</type>...
  </outputParam>
  <outputParam>
    <name>POOrder</name>
    <type>PurchaseOrderType</type>...
  </outputParam>
  <externalInteraction>manual</externalInteraction>
</decomposition>

...

<task id="Create_Purchase_Order_104">
  <name>Create Purchase Order</name>
  <flowsInto>
    <nextElementRef id="Approve_Purchase_Order_1901" />
  </flowsInto>
  ...
  <completedMappings>
    <mapping>
      <expression query=
        "&lt ;POOrder&gt ;{ / Create_Purchase_Order / POOrder/* }&lt ; / POOrder&gt ;" />
      <mapsTo>POOrder</mapsTo>
    </mapping>
  </completedMappings>
</task>
```

```

    </mapping>
    ...
</completedMappings>
<resourcing>
    <offer initiator="system">
        ...
    </resourcing>
    <decomposesTo id="Create_Purchase_Order" />
</task>

```

Listing 1 Conceptual information for task *Create Purchase Order*.

The XML extract in listing 2 represents the layout information for task *Create Purchase Order* and for its incoming and outgoing arcs. Layout information for a task includes the size and location of the task, as well as the path of the icon used to decorate the task. Layout information for an arc includes the location of the arc and a code for its line style, which can be orthogonal (encoded as 11), bezier (12) or spline (13).

```

<container id="Create_Purchase_Order_104">
    <vertex>
        <iconpath>.../Manual.png</iconpath>
        <attributes>
            <bounds x="177.0" y="80.0" w="32.0" h="32.0" />
        </attributes>
    </vertex>
    <label>
        <attributes>
            <bounds x="144.0" y="112.0" w="96.0" h="28.0" />
        </attributes>
    </label>
</container>

...

<flow source="InputCondition_16" target="Create_Purchase_Order_104">
    ...
    <attributes>
        <lineStyle>11</lineStyle>
        <points>
            <value x="188.5" y="31.0" />
            <value x="192.5" y="80.0" />
        </points>
    </attributes>
</flow>

...

```

```

<flow source="Create_Purchase_Order_104" target="Approve_Purchase_Order_1901">
  ...
  <attributes>
    <lineStyle>11</lineStyle>
    <points>
      <value x="192.5" y="111.0" />
      <value x="191.5" y="153.0" />
    </points>
  </attributes>
</flow>

```

Listing 2 Layout information for task *Create Purchase Order*.

7 Summary

This chapter introduced the main features of the YAWL Editor. It showed how to create and analyze the control logic of a YAWL process, how to specify the involved data and how to link organizational resources to process tasks.

Moreover the chapter demonstrated the error reporting feature of the Editor and provided an overview of the YAWL interchange format.

Exercises

1. Download the Editor from SourceForge (more details in the Chapter Notes) and start the tool. Open the Order Fulfillment example available with the distribution and browse the various sub-nets (cf. Appendix ??). Specifically, check how data and resourcing requirements have been specified and how different icons were used to identify the various task types.
2. Create a workflow specification with two atomic tasks: *Enter command* and *Display command*. Through the first task the user should be able to enter a sequence of characters into a variable *Command* of type *string*. The value of *Command* should then be displayed via the second task. Enrich the specification with icons from the *Task Icons* palette.
3. Extend the above example such that after displaying a command, the user should be asked to enter another command. However, if the user enters the command *quit*, the process should terminate straightaway without displaying the command *quit*.
4. Introduce a timer task *Timeout* which cancels task *Enter command* if no command is given within 20 seconds. In this case, ensure that the message *timeout has been*

triggered is visualized in task *Display command*. What happens to task *Timeout* if *Enter command* is completed in time?

5. Create a YAWL workflow specification which represents the following credit card application process:

A typical credit card application process begins when a customer submits an application with a proposed amount of credit. Next, a credit clerk verifies the status of the application. If the application is incomplete (e.g. the customer's credit history is missing) the clerk requests additional information and waits until the customer provides this information. However, if no information is received within a given period of time (e.g. 3 days), a request is sent again. This is done at most 3 times, then the process terminates assuming the customer is no longer interested in the application. If a complete application is received on time, the clerk verifies the customer's income and credit history. Different checks need to be performed depending on the amount of credit (e.g. more stringent requirements may apply to amounts greater than \$1000). After this, the application is passed on to a manager who decides whether to accept or reject it. In either case, the customer is notified of the decision and the process finishes.

6. Extend this workflow by allowing the possibility to cancel the application process at any time after a complete application is received but before the manager decides on the application.

7. Apply the following change to the credit card application process:

When an approved application is notified, the customer is asked to choose any extra features they may want to add. These include: customized card, reward program and secondary cardholder(s).

Chapter Notes

In this chapter we used the YAWL Editor version 2.0. The tool, along with the other components of the YAWL system, can be downloaded from the YAWL project Website hosted by SourceForge.³ Here the reader can find documentation on various aspects of the Editor and consult the mailing lists and forums to learn the latest news about the system.

The *YAWL user manual* is the complete reference documentation for the YAWL user. It contains information on how to install YAWL and dedicated guides for the Editor and the YAWL runtime environment. In addition, it includes the tutorial *Getting started with YAWL*, which provides a brief overview for the reader interested in experimenting with the Editor. On the other hand, the reader interested in advanced data aspects can consult the section *How to manipulate data in YAWL* of this manual.

³ sourceforge.net/projects/yawl

The development aspects behind the YAWL system are illustrated in the *Technical manual*. Finally, software bug reports and feature requests can be made through the Google code project.⁴

⁴ code.google.com/p/yawl